情報科学 2011　久野クラス　＃9

久野　靖*

2011.12.9

今回は、これまでお世話になってきた Ruby 実行系のような言語処理系の内部で、どのようなワザが活躍しているのかを見ていくことにします。具体的な内容としては次のものがあります。

- プログラミング言語処理系の内幕
- 木構造の続編 — 抽象構文木
- オブジェクト指向の続編 — 動的分配と継承

今回から B 課題はなくなりますが、何か自主的に出したい人は出して頂ければ Web に掲載してコメントをつけます (ただし点数はつけません)。では前回の演習問題解説から。

1　演習問題解説

1.1　演習 2 — エディタバッファのメソッド追加

この演習については、メソッドのみだけ掲載します。まず削除:

```
def delete
  if atend then return end
  @cur = @prev.next = @cur.next
end
```

これは前回授業でもかなり説明しましたが、要は (1) 最後の EOF は消さないようにする、(2)`@cur` は現在行の 1 つ先にする、(3)`@prev` の「次」も同じく現在行の 1 つ先にする、ということですね。どれかが足りないとおかしくなるので注意。次に交換です:

```
def exch
  if atend || @cur.next == @tail then return end
  a = @prev; b = @cur.next; c = @cur; d = @cur.next.next
  a.next = b; b.next = c; c.next = d; @cur = b
end
```

このように込み入ったつなぎ換えは、作業変数を使った方が間違えないで済みます。まず現在行か次の行が「おしまい」だったら交換できないのでそれを除外し、あとは最終的に並ぶ 4 つのセル (中央の 2 つが交換) を変数 `a`、`b`、`c`、`d` に入れて、つなぎ直し、`@cur` を変更します。

次は 1 つ戻るですが、せっかくある程度メソッドが作ってあるわけですから、「`@prev` を覚えておき、先頭に行ってから現在行が覚えておいた行になるまで 1 行ずつ進む」方法で作ってみました:

```
def backward
  if @prev == @head then return end
  a = @prev; top; while @cur != a do forward end
end
```

*筑波大学大学院経営システム科学専攻

全部反転はちょっと大変そうですね。要は、先頭から順にたどりながら、今まで「前→後」の対だったものを「後←前」の順になるように参照をつなぎ換える (ただし先頭と末尾はそれなりに対処)、という方針です:

```
def invert
  top; if atend then return end
  a = @cur; b = @cur.next; a.next = @tail
  while b != @tail do c = b.next; b.next = a; a = b; b = c end
  @head.next = a; top
end
```

先頭と末尾の対処はまず最初に先頭の次を`@tail`にし、最後にループを抜けてきた時のセルを先頭 (`@head`の次) にする、ということです。なお、バッファ内に 1 行しかない時はループ周回数が 0 で、その時もちゃんと動作することに注意。双方向リストの実現例は長くなりますし、やってみたい人のお楽しみなので省略させてください。

1.2 演習 4 — エディタの機能強化

まず、「指定した行へ行く」機能はバッファ側で「何行目」を管理するのがよいので、エディタバッファ全体を示します。基本的に、インスタンス変数`@lineno`を追加して、現在行が変化するメソッドではこれを更新します。`invert`みたいにぐちゃぐちゃにいじる場合は最後に`top`を呼ぶので、ここで 1 にリセットされるから考える必要はありません。そしてこれがあれば`backward`はもっと簡単になるので、これも直しました。行番号を間違いなく維持するのはきわどそうに見えますが、`@lineno`をアクセスするのもバッファ内容や現在位置を変更するのも`Buffer`の中だけなので、この中できちんと処理すれば大丈夫です。つまり、オブジェクト指向の持つ**カプセル化**の機能によって、プログラムが正しく構成し易くなっているわけです。

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
    @lineno = 1
  end
  def getlineno
    return @lineno
  end
  def goto(n)
    top; (n-1).times do forward end
  end
  def atend
    return @cur == @tail
  end
  def top
    @prev = @head; @cur = @head.next; @lineno = 1
  end
  def forward
    if atend then return end
    @prev = @cur; @cur = @cur.next; @lineno = @lineno + 1
  end
  def insert(s)
    @prev.next = Cell.new(s, @cur)
    @prev = @prev.next; @lineno = @lineno + 1
  end
  def print
```

```
      puts("  " + @cur.data)
    end
# delete、exch は上掲のとおり。backward は以下のように変更
    def backward
      goto(@lineno - 1)
    end
    def invert
      top; if atend then return end
      a = @cur; b = @cur.next; a.next = @tail
      while b != @tail do
        c = b.next; b.next = a; a = b; b = c
      end
      @head.next = a; top
    end
    def subst(str)
      if atend then return end
      a = str.split('/')
      @cur.data[Regexp.new(a[1])] = a[2]
    end
    def read(file)
      open(file, "r") do |f|
        f.each do |s| insert(s) end
      end
    end
    def save(file)
      top
      open(file, "w") do |f|
        while not atend do f.puts(@cur.data); forward end
      end
    end
  end
```

エディタドライバ側も一応示します (「位置変更しない指定行数プリント」も、最初に行番号を覚えて、印刷し終わったらそこに戻ればよいので簡単です):

```
  def edit
    e = Buffer.new
    while true do
      printf(">")
      line = gets; c = line[0..0]; s = line[1..-2]
      if c == "q"    then return
      elsif c == "t" then e.top; e.print
      elsif c == "p" then
        e.print; l = e.getlineno;
        s.to_i.times do e.forward; e.print end; e.goto(l)
      elsif c == "i" then e.insert(s)
      elsif c == "r" then e.read(s)
      elsif c == "w" then e.save(s)
      elsif c == "s" then e.subst(s); e.print
      elsif c == "d" then e.delete
      elsif c == "x" then e.exch
```

```
    elsif c == "b" then e.backward
    elsif c == "v" then e.invert
    elsif c == "a" then e.forward; e.insert(s); e.backward
    elsif c == "c" then e.delete; e.insert(s); e.backward
    elsif c == "g" then e.goto(s.to_i)
    else                e.forward; e.print
    end
  end
end
```

1.3 演習 6 — 表探索の改良と計測

ここではハッシュ表と呼ばれる手法を説明します。[1]その原理は次のとおりです:

- 整数キーの範囲ぶんの配列を用意できれば、ビンソートと同様に 1 発でそのキーの値を格納/参照できるから極めて速い。
- しかし整数キーの範囲が広いと領域が大きくなり実用的でない。
- そこで整数キーを適当に「折り畳んで」用意した配列の範囲に写像し、そこに格納すればよい。この関数を $hash1(k)$ と呼ぶ ことにする。
- ただし、複数のキーが配列の同じ位置に写像されることがある。[2]この時は、別の関数 $hash2(k)$ により 値 N を計算し、N 個先、そこもふさがっていたらさらに N 個先、…のように空いている場所が見つかるまで探していく。

この方式で実装した表のクラスを以下に示します:

```
class IntStrTable3
  def initialize
    @keys = Array.new(9973, -1); @vals = Array.new(9973, nil)
  end
  def hash1(n) return n % @keys.length end
  def hash2(n) return n % 2971 + 1 end
  def put(key, val)
    h1 = hash1(key); h2 = hash2(key)
    while true do
      if @keys[h1] == key then @vals[h1] = val; return end
      if @keys[h1]<0 then @keys[h1]=key;@vals[h1]=val; return end
      h1 = (h1 + h2) % @keys.length
    end
  end
  def get(key)
    h1 = hash1(key); h2 = hash2(key)
    while true do
      if @keys[h1] == key then return @vals[h1] end
      if @keys[h1] < 0 then return nil end
      h1 = (h1 + h2) % @keys.length
    end
  end
end
```

配列のサイズがヘンな数ですが、10000 に近い素数を選んだものです。素数を用いるのは、N 飛びに調べていく時に「同じところをぐるぐる調べてしまう」ことがないので (N とその素数の最大公約数が 1 だから)、空きがある限りその空き

[1]本文ではこの名前は出さずにヒントだけを説明していました。
[2]これを衝突 (collision) と呼びます。

が見つかることが保証されます。あとこれはちょっとさぼっていて、表が満杯になった時のチェックをしていなません。満杯になると、入れる場所が見つからなくて無限に探し始めてしまいます。改良するには、入れた個数を別に数えておいてチェックすればよいでしょう。

表 1: 表のさまざまな実装の計測

実装	1000	2000	3000	5000	10000
線形探索	0.3984375	1.578125	3.5625	9.578125	39.3671875
	0.78125	3.0546875	7.0390625	18.734375	79.515625
2 分探索木	0.03125	0.0703125	0.09375	0.1875	0.375
	0.015625	0.078125	0.078125	0.1484375	0.2890625
ハッシュ	0.0078125	0.015625	0.015625	0.0234375	0.09375
	0.0078125	0.0078125	0.015625	0.03125	0.296875
Ruby の	0.0	0.0	0.078125	0.015625	0.03125
Hash	0.0	0.0078125	0.0	0.078125	0.015625

計測結果を表 1 に示します。計測のしかたは本文と同じく、N 回の put、N 回の get の時間を計測し、キーは 0〜100000 のランダム。上段は N 個の put、下段は NN 個の get の時間です。なお、最後の Ruby の Hash は次のような簡単なクラスを作ってインタフェースを合わせて計測しました (だいぶ例題が長くなってきたので、1 行で済むメソッドは 1 行に書くようにしましたが、その場合、引数が無くても「()」を書く必要があるので注意してください):

```
class IntStrTable4
  def initialize() @hash = {} end
  def put(key, val) @hash[key] = val end
  def get(key) return @hash[key] end
end
```

これを見るとハッシュ表が圧倒的に速いことがよく分かります (Ruby の Hash も名前どおり、内部の実装はハッシュ表になっています)。ハッシュ表の場合、表が十分空いていれば、「1 発で」項目が検索できるため、項目当たりの検索時間は**定数時間** (constant time) すなわち $O(1)$ になります。

ところで、上で挙げたハッシュ表の実装で 10,000 個の場合が妙に遅いですが…これは、表のサイズが 9973 なので 10,000 回ランダムに値を投入すると表は (重複はあるにせよ) ほぼ満杯になり、そのため遅くなっているわけです。ハッシュ表を実用にする場合は、最初に十分大きさを見積もって、表の充填率が 50%以下になるように設計するのがよいとされています。

演習 8 — 2 分探索木の弱点

2 分探索木で挿入時にキーが順番になっていたらどうなるかを調べるために、benchtable の変形版を作ってみます:

```
def benchtable1(cnt, tbl)
  puts(bench(1) do cnt.times do |i| tbl.put(i, "") end end)
  puts(bench(1) do cnt.times do |i| tbl.get(i) end end)
end
```

これで計測してみましょう:

```
irb> benchtable1(1000, IntStrTable2.new)
0.8359375
0.6875
=> nil
irb> benchtable1(2000, IntStrTable2.new)
3.3515625
```

```
2.75
=> nil
irb> benchtable1(3000, IntStrTable2.new)
7.53125
6.1953125
=> nil
```

確かに、ひどく遅くなっています。これは、キーが順番だと「1 直線の」木ができてしまい、通常なら木の深さが $logN$ 程度、1 回当たりの検索に掛かる時間が $O(logN)$ のところが、木の深さが N、1 回当たりの検索に掛かる時間も $O(N)$ になってしまうためです。これを避ける代表的な方法としては、次のものが知られています:

- **AVL 木** (AVL tree) — 左右の木に入っているノード数が常にバランスするように制御しながら挿入していく方法。Adelson-Velsky と Landis によって発明されたことからその頭文字を取ってこう呼ばれています。
- **スプレー木** (splay tree) [3]— アクセスするごとにアクセスした値が木の根元に来るように木を変形することで、一時的に偏った形になることがあっても、多数のアクセス全体を通した平均では $O(logN)$ でアクセスできる方法。ちなみに、このような、多数の操作全体を通した計算量のことを**償却計算量** (amortized computational complexity) と呼びます。

スプレー木について一応次節に説明を載せておきますが、授業内容としては取り上げません。

1.4 スプレー木

スプレー木は上で説明したように、アクセス毎にアクセスした値を持つノード (ないし、その値が木の中に無い場合はそれに隣接した値を持つノード) が木の根元に来るように木を変形する方法です。つまりデータ構造としては普通の 2 分探索木と同じで、アルゴリズムだけが違っているということです。次にコードを示します:

```
class IntStrTable3
  Node = Struct.new(:key, :data, :left, :right)
  def initialize() @root = nil end
  def put(key, val)
    if @root == nil
      @root = Node.new(key, val, nil, nil); return
    end
    @root = splay(key, @root)
    if key == @root.key
      @root.data = val
    elsif key < @root.key
      @root.left = Node.new(key, val, @root.left, nil)
    else
      @root.right = Node.new(key, val, nil, @root.right)
    end
  end
  def get(key)
    if @root == nil then return nil end
    @root = splay(key, @root)
    if key == @root.key then return @root.data else return nil end
  end
  def splay(key, root)
    dummy = lmax = rmin = Node.new(0, 0, nil, nil)
    while true do
      if key == root.key then break end
```

[3]splay は「広げる」という意味の英語で、殺虫剤のエアゾールなどでおなじみのスプレー (spray) とは l と r が違っています (だから何?)。

```
      if key < root.key
        if (child = root.left) == nil then break end
        if key < child.key
          root.left = child.right; child.right = root
          root = child; child = root.left
          if (child = root.left) == nil then break end
        end
        rmin.left = root; rmin = root
      else
        if (child = root.right) == nil then break end
        if key > child.key
          root.right = child.left; child.left = root
          root = child; child =root.right
          if (child= root.right) == nil then break end
        end
        lmax.right = root; lmax = root
      end
      root = child
    end
    lmax.right = root.left; rmin.left = root.right
    root.left = dummy.right; root.right = dummy.left; return root
  end
end
```

上記の変形を行うメソッド splay が中心であり、put や get はまず目指すキーを指定して splay を呼び、その後で根元にあるノードに対して処理を行います。put では木が空のとき (@root が nil) を別扱いし、splay した後根元に目的の値が無ければ根元の右か左に新しいノードを挿入します。

splay のコードは木を上から順に変形して行く、**トップダウン** (top-down) 処理と呼ばれるアルゴリズムを用いています。全体方針として、dummy というノードをまず用意し、その左側には指定されたキーより大きいノードを、右側には指定されたキーより小さいノードをつないで行きます。変数 rmin と lmax はそれぞれ左側と右側の枝の先端を指していて、これらのそれぞれ左下と右下に次のノードをつなぎます。

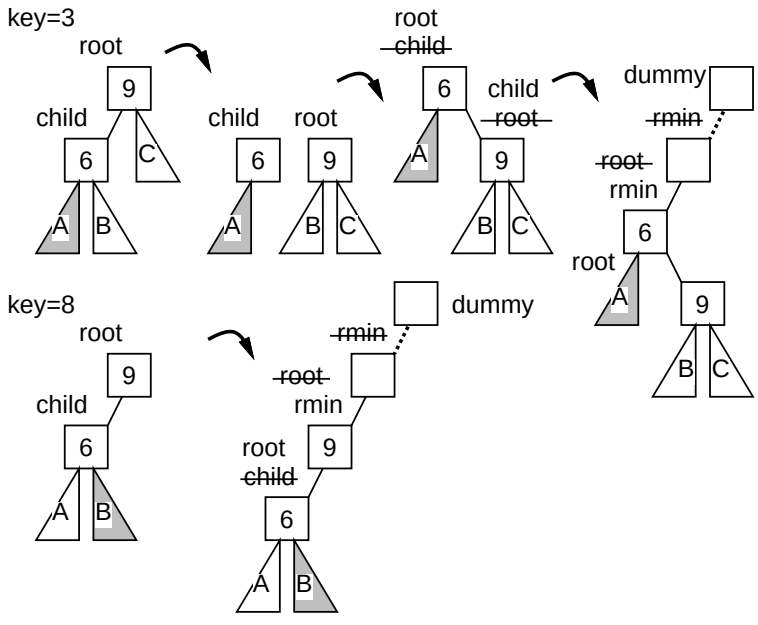

図 1: スプレー木のトップダウン処理

たとえば、現在 root が 9 のノードでその左に 6 のノードがあり、3 または 8 のノードを持ち上げるものとします (図 1)。9-6-3 は枝を左・左とたどることになりますが、その場合は中間のノード 6 が上に、上にあったノード 9 がその右下になるように木を変形します。この操作を**回転** (rotation) と呼びます。回転し終わったらその部分木を rmin の位置につないで、下に降りて行きます。一方 9-6-8 の場合は左・右とたどることになるので、回転はせずにそのままつないで下に降りて行きます。

この場合、6 は 8 より小さいのに dummy の左側につながれるのはヘンに思えますが、rmin は 9 の位置を指しているので、次に木が伸ばされる時に 9 の左下に伸ばされるので実質は 6 はつながれていないのと同じです。実際、6 は変数 root が指しているので次の周回で処理されて適切な位置 (今度は dummy の右側) につながれます。右・右や右・左とたどる場合も同様です。

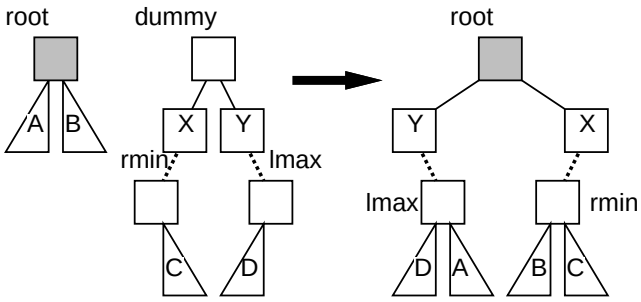

図 2: スプレー木のトップダウン処理の最終段階

このようにして木を下にたどって行き、目的の値が見つかるかそれ以上たどれなくなった時はループを抜け出し、その時点での root に入っているノードが根元になるように、図 2 の形に木を組み立てます。

なぜこうなるかというと、root の右下にあるもの (A) は、root より大きく、これまでにたどってきた全てのノードより小さい値が入っているので、lmin の左下にくっつけるのが正しく、左下 (B) も同様だからです。

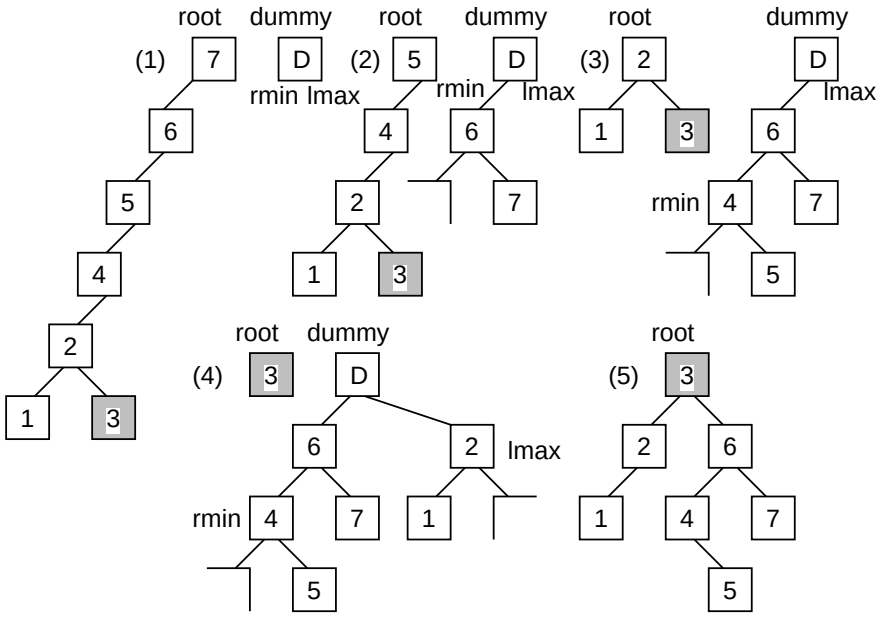

図 3: スプレー木のトップダウン処理の例

図 3 に比較的偏った木の下の方にある値を取り出す様子を示します。左・左の連続があると回転が行われてから rmin につながれるので、(1)〜(4) のように木の高さが低くなって行くことが分かります。

途中の状態では、先に説明したように rmin の左下や lmax の右下にくっついているものは一時的にくっついているだけなので、そのように図示しています。

最後に (5) で dummy の両下の値と root の値を組み合わせて木を完成させます。

2　抽象構文木とその実現

2.1　抽象構文木/式木

木構造の用途の 1 つとして、プログラミング言語による記述の構造を表す、というものがあります。たとえば「`x + 1`」という式は、`+` という (加算の) ノードの左右に `x`(変数ノード) と `1`(定数ノード) がぶら下がった木構造で表現できます (図 4 左)。このような、プログラミング言語の**構文** (syntax) に対応した木構造 (正確に言えばそれをさらに簡潔に整理したもの) を**抽象構文木** (abstract syntax tree)、その中で**式**に対応するようなものを**式木** (expression tree) と呼びます。

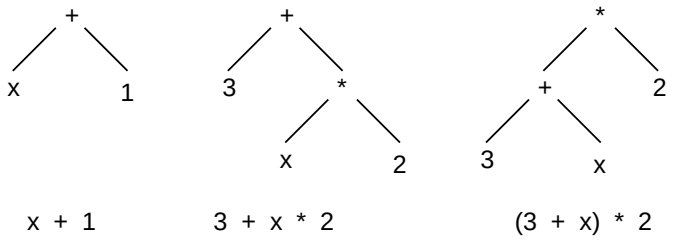

図 4: 演算式の抽象構文木 (式木)

もう少し複雑な「`3 + x * 2`」「`(3 + x) * 2`」を考えると、前者は x に 2 を掛けて 3 を足し、後者は x と 3 を足してから 2 を掛けることになります。これらの演算順序の違いも、木構造では素直に表せます (図 4 中・右)。

つまり、通常の数式では「乗除算は加減算より優先」「かっこ内は先に計算」などの規則がありますが、これらは「1 列に」式を表す時に必要な解釈の規則であって、式木になった状態では演算順序は木の構造によって表されています。

2.2　抽象構文木のオブジェクト表現/動的分配

抽象構文木をオブジェクトの集まりで表現してみましょう。前章の 2 分探索木では、各ノードをレコードで表現しましたが、こんどは「加算」「乗算」「変数」「定数」など各種のノードを 1 つのオブジェクトにするので、ノードの種別ごとに 1 つのクラスを用意します (変数の値はグローバルなハッシュ`$vars` に、変数名のエントリにその変数の値を格納する形で保持します。1 行目がその初期化です):

```
$vars = {}
class Add
  def initialize(l, r) @left = l; @right = r  end
  def exec() return @left.exec + @right.exec end
  def to_s() return '('+@left.to_s+' + '+@right.to_s+')' end
end
class Mul
  def initialize(l, r) @left = l; @right = r end
  def exec() return @left.exec * @right.exec end
  def to_s() return '('+@left.to_s+' * '+@right.to_s+')' end
end
class Lit
  def initialize(v) @left = v end
  def exec() return @left end
  def to_s() return @left.to_s end
end
class Var
  def initialize(v) @left = v end
  def exec() return $vars[@left] end
  def to_s() return @left.to_s end
end
```

クラス Add と Mul は加算と乗算のノードを表します。これらは作成時に左右の子供を受け取り、インスタンス変数@left と@right に格納します。メソッド exec は計算を実行して結果を返すもので、左と右それぞれの計算を実行した後、その結果を加算/乗算したものを返します。to_s は文字列表現を返すので、左の子と右の子の文字列表現を+や*でつないだものを返します。ただし、入れ子関係が分からなくなると困るので、全体をかっこで囲むようにしています。

クラス Lit と Var は定数と変数のノードを表現します。作成時に定数はその値、変数はその名前文字列を受け取り、インスタンス変数@left に格納します (各ノードで変数名をできるだけ揃えるようにしました)。exec は定数では値を返すだけですが、変数では$vars に格納されている値を取り出して返します。to_s は変数や定数の文字列を返すだけです。

ここで、@left.exec のようなメソッド呼び出しは、@left に現在何が入っているかに応じて、そのオブジェクトのメソッドを呼び出すことに注意してください。たとえば、@left が Add オブジェクトなら足し算、Var オブジェクトなら変数値の参照が行われて、その結果が返されます。

このように、「実行時に実際に使われているものに応じてメソッドが選択される」機能のことを**動的束縛** (dynamic binding) ないし**多態** (polymorphism) と言い、オブジェクト指向言語に不可欠な機能です。このおかげで、式の中身が何であっても「実行する」とだけ言えばその内容に応じて動作してくれるわけです。

では実際に動作させてみましょう。

```
irb> $vars['x'] = 5
=> 5
irb> e = Add.new(Var.new('x'), Lit.new(1))
=> ...
irb> puts(e, e.exec)
(x + 1)
6
=> nil
irb> f = Add.new(Lit.new(3), Mul.new(Var.new('x'), Lit.new(2)))
=> ...
irb> puts(f, f.exec)
(3 + (x * 2))
13
=> nil
irb> g = Mul.new(Add.new(Var.new('x'), Lit.new(3)), Lit.new(2))
=> ...
irb> puts(g, g.exec)
((x + 3) * 2)
16
```

これは、変数 x の値を 5 とし、あとは図 4 の 3 つの式木を組み立て、それを表示した後、実行して結果を打ち出しているものです。確かに正しく実行 (計算) できていますね。

演習 1 上の例題を打ち込み、式の内容は変えて、正しく計算できることを確認せよ。できたら、以下のようなノードのクラスを追加し、動作を確認せよ (to_s での表現方法は適当に決めてよい)。

a. 引き算のノード Sub、除算のノード Div。

b. 代入 Assign。このノードを exec すると、左のノード (Var であることを前提) の文字列表現 s をキーとして配列$vars[$s$] に右のノードを exec した結果を書き込む。結果としては、書き込んだのと同じ値を返す。次のようになるはず。

```
irb> e = Assign.new(Var.new('x'), Add.new(Var.new('x'), Lit.new(1)))
=> ...
irb> e.exec
=> 6
irb> $vars['x']
=> 6              ← 5 だったのが 6 に増える
```

c. 反復 Loop。このノードを exec すると、左のノードを exec して得た回数だけ、右のノードを繰り返し exec する (結果は何を返してもよい)。次のようになるはず。

```
irb> f = Loop.new(Lit.new(20), e)  ← e は上で用意した x = x + 1
=> ...
irb> f.exec
=> ...
irb> $vars['x']
=> 26              ← 20 回 1 ふやしたので 26
```

d. 連続 Seq。このノードを exec すると、左の子ノード、右の子ノードの順に exec する。結果としては、右の子ノードを exec した結果を返すものとします。次のようになるはず。

```
irb> g = Seq.new(f, f)  ← f は上で用意した 20 回の x = x + 1
=> ...
irb> g.exec
=> 46
irb> $vars['x']
=> 46              ← 20 回ふやすことを 2 回連続したので 46
```

3 抽象構文木によるプログラムの表現

3.1 継承によるくくり出し

前節の例題を見て、Add と Mul のコードはとても似ていると思いませんでしたか。実は、クラスを沢山作り始めると、しばしばこのようなことが起きます。このような時、クラス方式のオブジェクト指向言語では**継承** (inheritance) と呼ばれる機能を使うことで、記述を簡潔にできます。継承とは次のような機能を言います:

- クラスを定義する時、既にある別のクラスを土台として新しいクラスを定義できる。土台にするクラスを**親クラス** (parent class) ないし**スーパクラス** (superclass)、新しく作るクラスの方を**子クラス** (child class) ないし**サブクラス** (subclass) と呼ぶ。
- 子クラスは親クラスのインスタンス変数、メソッド定義をそっくりそのまま引き継ぐ (「継承」の意味)。
- 子クラスで独自のインスタンス変数やメソッド定義を追加できる。
- 子クラスで親クラスと同名のメソッドを定義することで、親クラスのメソッドを差し換えることができる。これを**オーバライド** (override) と呼ぶ。

先のコードを継承を用いて構造を整理してみましょう。まず、全ノードの土台となるクラス Node を用意し、共通のメソッド (ここでは to_s だけですが) を用意します:

```
$vars = {}
class Node
  def initialize(l=nil, r=nil)
   @left = l; @right = r; @op = '?'
  end
  def to_s()
    return '(' + @left.to_s + @op.to_s + @right.to_s + ')'
  end
end
```

@left、@right が左と右の子というのは同じですが、さらに表示用の**演算子** (operator — +、/などの演算記号のこと) を入れるインスタンス変数@op も追加しました。

これを土台に Add をサブクラスとして定義します (Ruby では「< 親クラス」という指定があるとそのクラスが指定した親クラスのサブクラスとなります):

```
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
```

このサブクラスでは initialize と exec だけを定義していて、前者は親クラスにあるメソッドなのでオーバライド (差し換え) になります。initialize の最初にある **super** というのは、親クラスの同名のメソッドを (同じ引数で) 呼び出すという意味になり、これにより@left、@right の初期化が正しく行えます。もし引数を変更したい場合は、(...) で独自の引数を指定することもできます。@op については'+' を自分で入れています。exec についてはこれまでの (継承を使わない) 版と同じですね。以下同様にして、他のノードも定義してみましょう:

```
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def exec() return @left.exec - @right.exec end
end
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def exec() return @left.exec * @right.exec end
end
class Div < Node
  def initialize(l, r) super; @op = '/' end
  def exec() return @left.exec / @right.exec end
end
class Lit < Node
  def exec() return @left end
  def to_s() return @left.to_s end
end
class Var < Node
  def exec() return $vars[@left] end
  def to_s() return @left.to_s end
end
```

Lit と Node については、自前で to_s を定義することにしたので、initiailze はオーバライドしていません (@op に固有のものを入れる必要がないため)。

3.2 制御構造の実現

もっと違った動作のノードとして、「代入」「連接」「ループ」「何もしない」を用意しました。代入は、左辺が変数であるものとして、その@left に変数文字列が入っているはずですから、$vars のその変数名のエントリに右辺の値を格納します:

```
class Assign < Node
  def initialize(l, r) super; @op = '=' end
  def exec() v = @right.exec; $vars[@left.to_s] = v; return v end
end
```

連接は、左を実行して、それから右を実行し、その結果を返すだけです。:

```
class Seq < Node
  def initialize(l, r) super; @op = ';' end
  def exec() @left.exec; return @right.exec end
end
```

ループは左を実行した結果の回数だけ右を繰り返し実行します。最後に実行した値を返すため変数 v に毎回値を保存しています (0 回実行の場合は 0 が返ります):

```
class Loop < Node
  def initialize(l, r) super; @op = 'L' end
  def exec()
    v=0; @left.exec.times do v=@right.exec end; return v
  end
end
```

「何もしない」というのは後で使うものですが、to_s もオーバライドしてただの「?」を返すようにしてあります。:

```
class Noop < Node
  def exec() return 0 end
  def to_s() return '?' end
end
```

ではちょっと試してみましょう。以下は、n に 5 を入れ、x に 1 を入れ、n の回数だけ繰り返し、x = x * n と n = n - 1 を実行し、最後に x の値を全体の結果とする (つまり 5 の階乗を計算する) コードの抽象構文木を組み立て、印刷して、実行します:

```
def test1
  e =
    Seq.new(
      Assign.new(Var.new('n'), Lit.new(5)),
      Seq.new(
        Assign.new(Var.new('x'), Lit.new(1)),
        Seq.new(
          Loop.new(
            Var.new('n'),
            Seq.new(
              Assign.new(Var.new('x'), Mul.new(Var.new('x'),
                         Var.new('n'))),
              Assign.new(Var.new('n'), Sub.new(Var.new('n'),
                         Lit.new(1))))),
          Var.new('x'))))
  puts(e)
  return e.exec
end
```

では実行してみましょう:

```
irb> test1
((n=5);((x=1);((nL((x=(x*n));(n=(n-1))));x)))
=> 120
```

確かに、ちゃんと階乗が計算できています。表示はちょっと読みにくいですが…

このように、プログラムの構造を内部的に表現し、それを「実行」することで、「プログラムの記述どおりの動作を行うプログラム」が作れます。一般に「プログラムの記述どおりの動作を行うプログラム」のことを**インタプリタ** (interpreter — 解釈実行系) と呼びます。

ここで示したような抽象構文木を直接解釈するタイプのインタプリタは作りやすく、それなりに使われています (ただし遅いという弱点があります)。たとえば Ruby のバージョン 1.8.x までの実行系はこのタイプのインタプリタを用いています。

演習 2 ノードの種別として次のようなものを増やしてみよ。

a. 大小比較の演算子。ここでは値を全部整数としているので、たとえば「x < y」は条件の成否に応じて 1 または 0 を結果として持つ、ということにするとよい。

b. while 文。条件部分は「0 が false、それ以外はすべて true」として扱うものとする。

c. if 文。とりあえず then 部だけでよいが、頑張って else 部も書けるようにしたければそうしてもよい。

d. 絶対値、2 数の最大、最小などの演算子。

e. その他、目新しい/面白い機能を持った文や演算。

4 字句解析と構文解析

4.1 字句解析器

ここまでは、Ruby の構文を使って直接オブジェクト群を組み合わせ、抽象構文木を組み立ててきました。しかしもちろん、普段我々がプログラムを作る時は、普通にコードを書いて、言語処理系がそれを解析して抽象構文木を組み立てています。以下ではその簡易版を作ってみましょう。

プログラミング言語処理系の入口となるのは、入力を「名前」「定数」「記号」などに切り分ける**字句解析器** (lexical analyzer) と呼ばれる部分です。ここでは簡単のため「プログラムは 1 行だけ、すべての名前、定数、記号は 1 文字だけ、余分な空白などは一切あってはいけない」という非常に制限された字句解析用クラスを作りました:

```
class Lexer
  def initialize(s) @str = s + '$'; @pos = 0 end
  def peek() return @str[@pos..@pos] end
  def fwd() if @pos<@str.length-1 then @pos=@pos+1 end end
  def to_s() return @str[0..@pos-1]+'!'+ @str[@pos..-1] end
end
```

initalize は解析用の文字列を@str、解析位置を@pos に設定します。「$」は終わりの印の文字として使うもので、プログラム中には存在しないものとします。peek は「現在位置の文字を返す」メソッド、fwd は「現在位置を (終わりでない場合に)1 つ進める」メソッドです。最後に to_s はエラー表示用に「どこを読んでいるか」が分かりやすいように表現した文字列を返すようにしました。

動かしてみましょう:

```
irb> sc = Lexer.new('x=x+1')
=> #<Lexer:0x810b460 @str="x=x+1$", @pos=0>
irb> sc.peek
=> "x"
irb> sc.fwd
=> 1
irb> sc.peek
=> "="
irb> sc.fwd
=> 2
irb> sc.fwd
=> 3
irb> sc.fwd
=> 4
irb> sc.peek
=> "1"
irb> sc.fwd
=> 5
irb> sc.peek
=> "$"
irb> puts(sc)
x=x+1!$
=> nil
```

最後のは、「x=x+1 というプログラムを読み終わってファイル終端のところにいる」ということを表しているわけです。

演習 3 Lexer のインタフェース (メソッド peek、fwd を使うこと) は変えずに、普通のプログラミング言語のように複数文字から成る名前や定数 (整数だけでよい) を扱い、空白や改行は無視するような字句解析クラスを作り、後に出てくる構文解析器と組み合わせて動作させてみよ。[4]

4.2 BNF による構文定義

Ruby の構文もちゃんと説明していないのに恐縮ですが、構文が決まらないと解析器が作れないので、「おもちゃ言語」の構文を定義します:

$$
\begin{aligned}
&\mathit{prog} ::= \mathit{stat} \mid \mathit{stat}\ \texttt{';'}\ \mathit{prog}\\
&\mathit{stat} ::= \texttt{'\{'}\ \mathit{prog}\ \texttt{'\}'} \mid \texttt{'L'}\ \mathit{expr}\ \mathit{stat} \mid \mathit{expr}\\
&\mathit{expr} ::= \mathit{fact} \mid \mathit{fact}\ \texttt{'+'}\ \mathit{expr} \mid \mathit{fact}\ \texttt{'-'}\ \mathit{expr}\\
&\quad \mid \mathit{fact}\ \texttt{'*'}\ \mathit{expr} \mid \mathit{fact}\ \texttt{'/'}\ \mathit{expr} \mid \mathit{fact}\ \texttt{'='}\ \mathit{expr}\\
&\mathit{fact} ::= \mathit{identfier} \mid \mathit{number} \mid \texttt{'('}\ \mathit{expr}\ \texttt{')'}
\end{aligned}
$$

この記法は **BNF**(Backus Normal Form) と呼ばれ、::= の左辺の記号を右辺の記号列で定義しています (「|」は「または」を表しています)。

上の文法では**識別子** (identifier — 名前のこと) と**数値** (number) が定義されていませんが、これらの構造は字句解析器のほうで決めるのが普通です。ここでは上記の字句解析器を使うので、それぞれ英小文字 1 文字、数字 1 文字に対応させます。

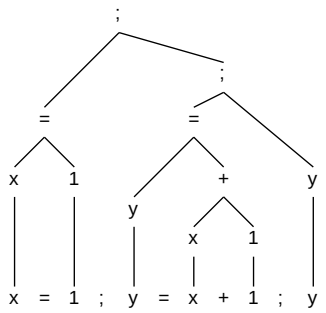

図 5: おもちゃ言語の抽象構文木

図 5 に「x = 1; L 5 x = x * 2; x」という簡単なプログラム (2 を 5 回掛ける、つまり 2 の 5 乗を計算するものです) に対する抽象構文木を示します。上掲の文法との対応関係を確認してみてください。

演習 4 次のプログラムに対する抽象構文木を描け。

```
a.  x=y=z=1
b.  x=2*3+4*5
c.  n=5;x=1;L(n){x=x*n;n=n-1};x
```

演習 5 自分が (演習 3 などで) 導入した独自の機能の構文を決めてそれを含むように「おもちゃ言語」の BNF を拡張せよ。さらに、その構文を持つプログラム例を選び、抽象構文木を描け。

4.3 再帰下降型構文解析器

字句解析を下請けに使ってプログラムを解析し、抽象構文木を組み立てるプログラムのことを**構文解析器** (syntax analyzer) または**パーザ** (parser) と呼びます。ここでは、**再帰下降解析器** (recursive descent parser) と呼ばれる方式の解析器を作ってみます。再帰下降解析器は、再帰手続きの動作がプログラムの構文木をたどる動作と類似していることを利用した構文解析方式です。その作り方は次のとおりです:

[4] 文字列を渡す代わりにファイルからプログラムを読むようにできると、なおよいでしょう。

- 各構文記号の定義ごとに 1 つずつ手続きを用意する。(手続きの引数は字句解析器で、返値はその手続きに対応する抽象構文木となります。)
- 各手続きは、その構文定義の右辺にある構文記号に対応する手続きを 1 つずつ呼び、返されてきた木を束ねて自分の構文木を組み立てる。(呼ぶものが 1 つしか無い場合は、返された木をそのまま返すこともあります。)
- 右辺に現れるもののうち個別の文字 (記号、英字、数字) については、次の文字がその文字であることを確認して入力を先に進める。
- 右辺が | によって複数の可能性から選択になっている場合は、どの選択肢へ進むかも入力に基づいて決定する。

特に重要なのは「入力に基づいて決定する」というところで、そのような選択がうまく行えない文法はこの方法では解析できなません。幸い、先にに出てきた文法はこの方法で解析できるようになっています。というかもちろん私がそのように設計したのですが。

図 6 に、おもちゃ言語の簡単なプログラムを再帰下降解析によって解析している様子を図示します。一番先頭の *prog* はどの規則でも先頭が *stat* なので *stat* を呼び出します。*stat* は次の文字が「{」でも「L でもないので *expr* を呼び出し、*expr* はどの規則でも最初は *fact* なので *fact* を呼び出し、*fact* の中では次の文字が'x' なのでこれを変数として認識します。このように、構文木の構造と再帰手続き群の呼び出し関係がきっちり対応していることが見て取れます。

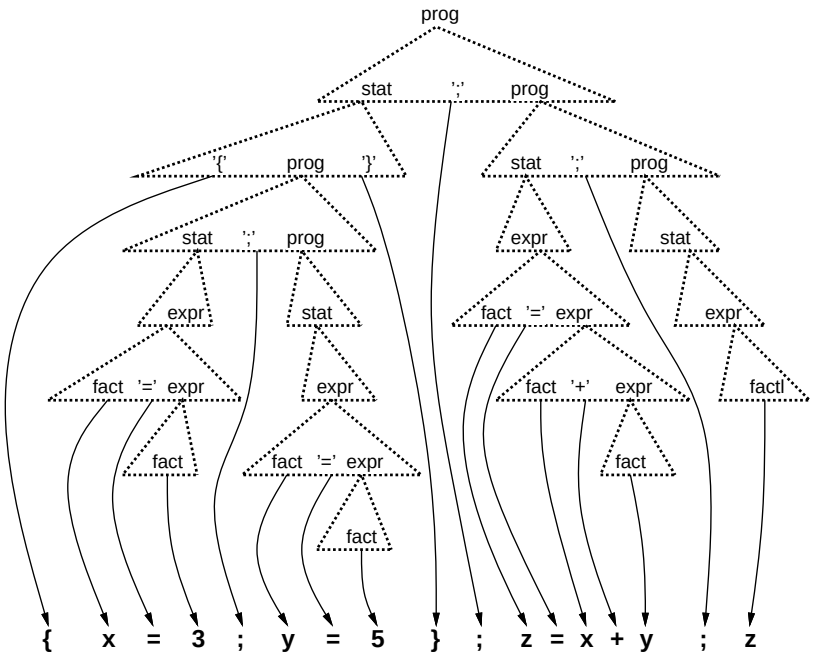

図 6: 再帰下降解析の呼び出し例

構文記号ごとに対応する手続きを見ていきましょう。まず *prog*:

```
def prog(sc)
  s = stat(sc); c = sc.peek
  if c == '$' || c == '}' then return s
  elsif c == ';' then sc.fwd; return Seq.new(s, prog(sc))
  else puts('STAT:' + sc.to_s); return Noop.new
  end
end
```

prog の右辺の先頭はいずれにせよ *stat* なので、まず stat を呼びます。次の文字が「$」か「}」ならこれで *prog* は終わりなので (なぜそうなのかは長くなるので省略します。興味ある人は参考文献を見てください)、*prog* ::= *stat* が適用されて *stat* の木をそのまま返します。次の文字が「;」の場合は *prog* ::= *stat* ';' *prog* なので、「;」は読み捨てて prog を再帰呼び出しし、その結果の木と先の *stat* の木とを Seq ノードで連結したものを返します。どちらでもない場合は構文エラーなのでエラーメッセージを出力した後「何もしないノード」を返します (クラス None は実はそのために用意したのでした)。

次に *stat*:

```
def stat(sc)
  c = sc.peek
  if c == '{' then
    sc.fwd; p = prog(sc)
    if sc.peek != '}'
      puts('NO_}:' + sc.to_s); return Noop.new
    end
    sc.fwd; return p
  elsif c == 'L'
    sc.fwd; e = expr(sc); return Loop.new(e, stat(sc))
  else
    return expr(sc)
  end
end
```

まず先頭が「{」である場合は $stat ::='\{'prog'\}'$ に対応するので、まず 1 文字進め、prog を呼び、次が「}」であることを確認して (そうでない場合はエラー)、prog から返された木をそのまま返します。先頭が「L」の場合は $stat ::= 'L'expr\ stat$ に対応するので、1 文字進め、expr と prog を呼んでこれらを子にもつ Loop ノードを作って返します。それ以外の場合は $stat ::= expr$ に対応するので、expr を呼んでその結果をそのまま返します。

次は *expr* を見てみましょう:

```
def expr(sc)
  e = fact(sc); c = sc.peek
  if c == '+' then sc.fwd; return Add.new(e, expr(sc))
  elsif c == '-' then sc.fwd; return Sub.new(e, expr(sc))
  elsif c == '*' then sc.fwd; return Mul.new(e, expr(sc))
  elsif c == '/' then sc.fwd; return Div.new(e, expr(sc))
  elsif c == '=' then sc.fwd; return Assign.new(e, expr(sc))
  else return e
  end
end
```

どれを選ぶにせよまず最初は $fact$ なので fact を呼び、次に演算子のいずれかがあれば 1 文字進めて expr を呼び、これらを 2 つの子供として持つ適切な演算ノードを作って返します。これら以外の場合は $expr ::= fact$ に対応するので、fact の結果をそのまま返します。

最後は $fact$ です:

```
def fact(sc)
  c = sc.peek; sc.fwd
  if c >= 'a' && c <= 'z'
    return Var.new(c)
  elsif c >= '0' && c <= '9'
    return Lit.new(c.to_i)
  elsif c == '('
    e = expr(sc)
    if sc.peek != ')'
      puts('NO_):' + sc.to_s); return Noop.new
    end
    sc.fwd; return e
  else
    puts('FACTOR:' + sc.to_s); return Noop.new
```

```
    end
  end
```

次の文字が英小文字なら変数ノード、数字なら定数ノードを作って返します。「(」なら $fact ::= '('expr')'$ なので expr を呼んでその結果を返しますが、ただし対応する「)」がなければエラーとします。これらのどれでもない場合はやはりエラーです。以上です。4 つの再帰手続きがあって分かりにくいかもしれませんが、要はそれぞれの手続きが構文記号に対応していて、構文規則にしたがって呼び出しを進めていく、ということですね。最後に 1 行プログラムを受け取って解析し実行するメソッドを作ります。

```
def run(s)
  e = prog(s); puts(e); puts(e.exec)
end
```

これを使って、5 の階乗を計算させたようすは次の通り。

```
irb> run 'n=5;x=1;Ln{x=x*n;n=n-1};x'
((n=5);((x=1);((nL((x=(x*n));(n=(n-1))));x)))
120
=> nil
```

ここでは「おもちゃの言語」でしたが、実際の言語処理系も基本的にはこのような原理でできています。ただし、最終的に実行するやり方が抽象構文木を解釈するインタプリタであるものの他に、マシン語を生成して実行させる**コンパイラ** (compiler) 方式だったり、マシン語よりは上位のレベルだが中間コードを生成してそれを解釈する**コンパイラインタプリタ** (comiler-interpreter) 方式だったりします。また、近年のコンパイラではコードの性能を向上させるように各種の変形を行う**最適化** (optimization) に多くの時間を割くようになっています。

なお、今回の処理系は構文さえ合っていれば意味がおかしいプログラムでも実行開始してしまいます。[5]実際の処理系では構文解析の後に**意味解析器** (semantic analyzer) と呼ばれるフェーズがあり、意味的なチェックをおこなっておかしいところがあれば実行に進まないようにします。

演習 6 上の解析+実行プログラムをコピーできるように用意するので、これを持って来て「5 の階乗」を計算してみよ。うまく行ったら、次の計算をするプログラムを打ち込んで動かせ (なぜ 9 かというと、数字が 1 文字でしか指定できないから。10 を指定したければ 5+5 とかすれば済みますが)。

a. 2 の 9 乗を計算する。
b. 9 番目のフィボナッチ数列の値を計算する。
c. ${}_9C_5$ を計算する。

演習 7 ここで示した構文ではすべての演算子の順位 (優先度) が同じであり、また演算子が**右結合** (right associative) である。[6]これを、乗除算が加減算に優先され、なおかつ左結合になるように手直ししてみよ。

演習 8 面白い言語を設計し、その処理系 (字句解析+構文解析+インタプリタ) を作成して実際にプログラムを書き、面白さについて考察せよ。何が面白いかは各自の判断に任されるものとする。

A　本日の課題 9A

「演習 1」〜「演習 2」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 9A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. プログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

Q1. 抽象構文木という考え方を理解しましたか。
Q2. 「動的分配」「継承」について納得しましたか。
Q3. 本日の全体的な感想と今後の要望をお書きください。

[5]たとえば「(x+1)=10」みたいなものでもエラーにならずに動作します。その結果がどうなるのかは考えてみてください。
[6]右結合とは、x - y - 1 のように演算子が並んでいる時、x - (y - 1) のように右側の演算が優先されることを 言います。これに対し、普通の計算規則は**左結合** (left associative) であり、左側の演算が優先されます。